Multiprocessors and Linux

Krzysztof Lichota
lichota@mimuw.edu.pl

Rationale for going multiprocessor

- Single processor speed cannot be increased quickly enough for processing needs
- The tendency is to multiply processing power by increasing number of processors/cores
- SMP is long available in Intel and other processors, there was attempt for simulated SMP (HyperThreading), current standard is 2 processor cores, 4 is available on the consumer market, 8 will be available soon

Scaling even more

- SMP scales up to 8, maybe 16 processors
- Above 16 processors it is hard to achieve good performance due to various bottlenecks (for example maintaining cache coherency)
- One possible solution is assymetric multiprocessing – specialized, separate computing subprocessor (for example Cell)
- Another is using NUMA

NUMA

- NUMA (Non-Uniform Memory Access) is a system in which processor group (node) has fast access to some memory (local memory), while access to other nodes memory (remote) is slow
- Similarly, node can have separate, local I/O devices or there can be I/O nodes
- Distance between nodes can be different (in various metrics)
- Operating system must be aware of properties of NUMA machine for efficiency

Linux kernel history of multiprocessor support

- 2.0 (June 1996) - SMP support using big kernel lock (BKL) for getting into kernel (only one processor could enter kernel) – 16 CPUs supported
- 2.2 (January 1999) – introduced fine-grained locking (spinlocks), removed big kernel lock in some places
- 2.4.0 (January 2001) – more fine-grained locking, especially in I/O subsystems, up to 64 CPUs
- 2.6 (December 2003) – NUMA and RCU support, up to 512 processors (IA64)

SMP

SMP problems

- Shared structures require synchronization between processors
- Locks cause contention and performance degradation when other CPUs must wait to enter critical section
- Cache thrashing should be avoided as it spoils performance significantly – processors should avoid using the same memory
- Interrupts should be balanced between processors to not become bottleneck

SMP problems (2)

- Some architectures require explicit synchronization of CPU caches using memory barriers
- Scheduler must be aware of system configuration to schedule tasks on the same CPU (to avoid cold cache), and to spread tasks in most efficient way in case of shared CPU resources (e.g. HT)

Spinlocks

- Classic, simple approach to locking, uses active waiting and atomic operations (e.g. compare-exchange or similar, DEC on x86)
- Memory bus must be locked during atomic comparison causing degradation in memory access for other processors
- Spinlock is in shared memory for all processor causing cache thrashing and expensive SMP operations (barriers)
- Code in Linux, Documentation

RW-spinlocks

- Spinlocks for readers-writers
- No lock contention for readers
- Used when writers are rare - have higher cost than normal spinlocks when there is contention
- Still the same problem with shared memory access on SMP
- Code

Atomic operations

- Atomic operations (atomic inc, atomic add, etc.) have smaller cost (i.e. processor cycles) than spinlocks
- Atomic operations do not cause contention
- Still they cause the same cache thrashing and shared memory problems on SMP

Refcounting

- Using atomic operations some locks can be removed by using reference counters for structures
- When structure is used, its counter is >0
- When structure is in use we can skip some locking
- Example: i_count in struct inode

Per-CPU data

- The best way to avoid cache thrashing and coherency problems is to use per-CPU data, which is not used (or rarely used) by other CPUs
- Example: per-cpu runqueues in 2.6 kernel scheduler

RCU

- Lock-free approach for reader-writer synchronization
- RCU stands for Read-Copy-Update
- Readers get reference to some structure and can use it freely, then release reference
- Writers copy structure which they want to modify, update it and insert into structure so that new readers get reference to updated structure
- When all readers using previous version of structure are gone, it can be freed

RCU (2)

- Freeing of old versions is deferred to quiescence period (all processors have left kernel mode or done other blocking operation, so no readers are present)
- Allows to avoid atomic operations, memory barriers and cache misses for readers
- Example (struct file usage): getting reference to struct file, freeing struct file

Memory barriers

- CPUs can introduce out-of-order execution of instructions
- CPUs can perform speculative fetching and execution of instructions
- CPU caches can synchronize with main memory and other caches in different moments on some architectures
- This can cause problems on SMP systems if we assume that write to some memory location is complete, but other processor does not see it yet

Memory barriers (2)

- Compilers also can rearrange instructions in code if they do not see dependencies and they think they optimize execution
- To force consistent state of memory on CPUs memory barrier must be used
- Barriers can guarantee that all LOAD operations were performed before LOAD operation after the barrier (read barriers), STORE operations were performed similarly (write barriers) or they can be combined (general barriers)

Memory barriers (3)

- Performing barrier on one processor makes sense only if the other processor also performs barrier – otherwise there is no guarantee
- Read barrier must be paired with write barrier

Memory barrier example

Code (CPU1), A=0, B = 0

- STORE A=1
- STORE B=1

Code (CPU2)

- LOAD A
- LOAD B

- We expect that if B == 1, then A == 1, while it might be also 0 due to reordering
  - STORE B=1 (CPU1)
  - LOAD B (CPU2)
  - LOAD A (CPU2)
  - STORE A=1 (CPU1)

Memory barrier example (2)

Code (CPU1), A=0, B = 0

- STORE A=1
- WMB
- STORE B=1

Code (CPU2)

- LOAD A
- RMB
- LOAD B

- Now all STOREs must be completed before STORE B=1 is executed, and all LOADs must be completed before LOAD B, so if B==1, then A==1

Memory barriers in Linux

- On Linux memory barriers are defined per-architecture as smp_rmb() (read barrier), smp_wmb() (write barrier) and smp_mb() (general barrier)
- Example: blk_trace_startstop() function uses smp_mb() to force blktrace_seq++ before setting trace_state = Blktrace_running, so that sequence numbers do not overlap in case of reordering

NUMA

NUMA and SMP shared issues

- Scheduling processes on nodes where their memory resides (CPU affinity, not critical on SMP)
- CPU hotplugging
- Discontiguous memory (as each node has separate memory), not critical on SMP
- Memory hotplugging

NUMA systems issues

- Placing exec-ed processes on least loaded nodes
- Avoiding accesses to remote memory
- Binding processes and memory to I/O devices

NUMA and HyperThreading

- HyperThreading introduces asymmetry in CPUs similar as in NUMA
- Some processors are not equal as they are “siblings” of the same physical CPU
- Example: 2 CPUs each with HyperThreading are seen as 4 CPUs, but it is better to schedule processes on different physical processors
- Multi-core processors also share some resources (like cache)
- Information about siblings in /proc/cpuinfo

CPUsets

- Linux supports defining hierarchy of processors and their memory using CPUsets
- Tasks can be limited to run only on specific nodes (scheduler takes care of migration)
- Memory can be assigned only on specific nodes and spread over nodes (VM subsystem allocates memory from specific node)
- Hierarchy is defined by user using cpuset pseudo-filesystem (similar to /proc)
- Interface in cpuset.h header file
- Linux kernel documentation about CPUsets

Linux scheduler and NUMA

- Separate runqueue for each processor
- Periodical rebalancing among processors
- HyperThreading and NUMA aware
- Linux scheduler can take into account nodes hierarchy, load, priorities and other factors
- See struct sched_domain, sched_balance_self()

Memory binding to processes

- Linux kernel supports binding memory processes to NUMA nodes since version 2.6.7
- Userspace processes can use mbind() and set_mempolicy() system calls to set from which nodes memory should be allocated
- Interface is in mempolicy.h header

Bibliography

- Columbia - supercomputer running Linux (
- http://www.ibm.com/developerworks/libra
- http://lse.sourceforge.net/numa/
- http://lse.sourceforge.net/numa/status/des
- http://www.gelato.org/pdf/may2005/gelato
- http://oss.sgi.com/projects/page_fault_perf
- http://www.linuxsymposium.org/proceedin
- http://en.wikipedia.org/wiki/RCU
- http://lxr.linux.no/linux+v2.6.23.5/Docume

Bibliography (2)

- http://en.wikipedia.org/wiki/Memory_barrie
- http://lxr.linux.no/linux+v2.6.23.5/Docume
- http://lxr.linux.no/linux+v2.6.23.5/Docume
- http://lxr.linux.no/linux+v2.6.23.5/Docume
- man mbind
- man set_mempolicy